Negative Effects of Bytecode Instrumentation on Java Source Code Coverage

Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy
Department of Software Engineering
University of Szeged, Szeged, Hungary
{dtengeri,hferenc,beszedes,gertom,gyimothy}@inf.u-szeged.hu

***Abstract*—Code coverage measurement is an important element in white-box testing, both in industrial practice and academic research. Other related areas are highly dependent on code coverage as well, including test case generation, test prioritization, fault localization, and others. Inaccuracies of a code coverage tool sometimes do not matter that much but in certain situations they can lead to serious confusion. For Java, the prevalent approach to code coverage measurement is to use *bytecode instrumentation* due to its various benefits over *source code instrumentation*. However, if the results are to be mapped back to source code this may lead to inaccuracies due to the differences between the two program representations. In this paper, we systematically investigate the amount of differences in the results of these two Java code coverage approaches, enumerate the possible reasons and discuss the implications on various applications. For this purpose, we relied on two widely used tools to represent the two approaches and a set of benchmark programs from the open source domain.**

***Index Terms*—Code coverage, white-box testing, Java bytecode instrumentation, source code instrumentation, coverage tools**

I. Introduction

Evolving software systems must rely on special testing activities, which are able to cope with constantly changing requirements and code. This includes, among others, white-box test design, massive regression testing, selective retesting, efficient fault detection and localization, as well as maintaining the efficiency and effectiveness of the test assets on the long term [1]. This paper investigates *code coverage*, a test completeness measure which plays an important role in many of the mentioned test activities. Essentially, code coverage indicates which code parts are excercised during the execution of a set of test cases on the system under test. The knowledge about the (non-)covered elements will underpin various decisions during these testing activities, so any inaccuracies in the measured data might be critical.

Software testers have long established the theory and practice of code coverage measurement: the various types of coverage criteria like statement, branch and others [2], as well as technical solutions including instrumentation for runtime analysis [3]. However, daily practice shows that the behavior of the tools that are available for this task is in many cases quite different than text-book methods.

In this paper, we concentrate on code coverage measurement for Java systems. In this area, there are peculiar issues which tool builders must face, the most notable one being how *code instrumentation* is done. This technique is used to place “probes” into the program, which will be activated upon runtime to collect the necessary information about the code coverage. In Java, there are two fundamentally different instrumentation approaches: *source code* level and *bytecode* level. Due to its various technical benefits (for example, it does not require the source code nor its recompilation, it can position the probes more precisely, and it can also handle generated code), bytecode instrumentation is preferred in many cases [4]. However, this approach also has its drawbacks compared to source code instrumentation, most notably the inability to perfectly map the coverage results back to the source code, which is essential in many applications. Interestingly, there has been very little research performed to systematically assess these drawbacks, while the benefits of bytecode instrumentation are more often emphasized. In our code coverage-related research, we also experienced significant differences in the two approaches, and oftentimes noticed inaccuracies due to the bytecode instrumentation with the tools we used. This motivated us to perform an empirical study to investigate the negative effects of bytecode instrumentation on the code coverage when it is to be observed on source code level. We did not want to deal with the *benefits* of bytecode instrumentation as it is much more well-known. Similar studies exist in relation to branches and statements [5], but we wanted to verify the differences on higher granularity, on method level. Also, we used real size systems with non-trivial test suites.

In this paper, we perform an empirical study involving two representative and popular code coverage tools, one from each instrumentation approach (JaCoCo and Clover), and a set of benchmark programs. Our goal was to assess the differences, most notably the inaccuracies of bytecode instrumentation. We did this by quantitative and qualitative analysis, and by elaborating on the possible impacts of the inaccuracies to a set of applications. We make the following main contributions:

- We extended the bytecode instrumentation tool JaCoCo to be able to provide per-test case coverage results. This enables fine-grained comparison of the results to the other tool, but will also enable other uses of the tool previously not possible.
- We manually verified the coverage results provided by the source code instrumentation tool Clover and found it correct. This will make possible to use this tool as a *ground truth* for source code coverage results.

- We performed a set of measurements to quantify the differences of the bytecode instrumentation approach with respect to source code. Although we used Java methods as code items and beforehand we expected only minor discrepancies due to this higher granularity, our findings indicate notable differences – as much as 24% of the actually covered items will be erroneously reported by the bytecode instrumentation tool – which may be critical in some situations.
- We list possible causes for the difference, which potentially enables using workarounds when bytecode instrumentation based approach is used.
- We elaborate on the impacts of the inaccuracies on applications such as white-box testing, fault localization, test selection, test case generation and mutation analysis.

The paper is organized as follows. Section II revisits various roles of code coverage independent of Java, and the possible impacts in different applications. In Section III, we discuss code coverage for Java and state our research aims. Section IV describes the basic setup for the experiments, the tools and the subject systems, while Section V presents the results of the empirical study. Section VI describes the threats to validity of the experiment, and finally, we conclude in Section VII.

II. Role of Code Coverage

The usual interpretation of the term "code coverage" is the amount of program code which is excercised during the execution of a set of test cases on the system under test. This indicator may be used simply as an overall "coverage percentage," but typically more detailed data is also available about individual program elements or test cases. Code coverage measurement is the basis of several software testing and quality assurance industrial practices, most notably *white-box* testing [6]. Apart from this significance, code coverage measurement is an actively investigated topic in academic research as well. Hence, the implications of inaccurate code coverage data may be multiple.

A. White-box testing

White-box testing (often referred to also as *structure-based testing*) is a dynamic test design technique that relies on code coverage to systematically verify the amount of tests needed to achieve a completeness goal (although, it is not clear whether coverage metrics themselves can predict the effectiveness of the test suite [7] or not [8]). This goal is sometimes expected to be complete (that is, 100%) coverage, however in practice this high level is rarely attainable due to various reasons. White-box testing is usually associated with lower testing levels such as unit tests, however there is no limitation to take advantage of code coverage on higher testing levels, too.

In white-box testing, code coverage measurement tools are often used that are able to report different kinds of overall coverage rates in terms of percentage, and also detailed reports about the covered and non-covered program elements. The inaccuracy of these tools may or may not notably influence the decisions to be made during the testing process, depending on the usage scenario and the nature of the inaccuracy, as summarized below:

- If the overall coverage percentage is required only (for example, as an initial assessment of the tests or to be reported in a summary report) then minor differences of a few percent are irrelevant.
- Generally, there may be two types of inaccuracies of a coverage tool. In the first case, it will erroneously report a not covered element for an actually covered code item. We treat this case the *safe but imprecise case* because it will draw the attention of the tester to a code for investigation superfluously, but will not miss important cases.
- On the other hand, if a piece of code is reported as covered while actually not being covered, this may lead to *false confidence* in the tests and potentially to higher risks. For example, the tester will not notice in this scenario that an important code item was not tested.

Note, that there are different levels of code coverage criteria (such as method, statement, decision, and many others), and the different tools may behave differently on these levels. However, the above reasoning will be generally valid for all different types of code coverage. Also, this is independent of a particular programming language and technology.

B. Other applications

Other applications of code coverage measurement include general software quality assessment [9], automatic test case generation [10], [11], code coverage-based fault localization [12], [13], test selection and prioritizaton [14], [15], [16], [17], mutation testing [18], [19], and in general program and test comprehension and traceability between the two [20]. As with white-box testing, the inaccuracy of code coverage measurement may affect these activities in different ways.

Certain applications do not suffer that much if the coverage data is not precise. This includes overall quality assessment, where the coverage ratio is typically used as part of a more complex set of metrics for software assessment. Here, a difference of a few percent usually does not affect the overall score. Program comprehension (and general project traceability) is supported by knowing which program code is executed by which test case. Depending on the usage scenario of this information, inaccurate results may lead to either false decisions or simply increased effort to interpret the data.

The other mentioned applications have high significance in academic research, and the accuracy and validity of the published results may be impacted by the issues with the code coverage data. In coverage-driven test case generation, for instance, the generation engine can be confused by an imprecise coverage tool because a falsely reported non-coverage will keep the generation algorithm trying to generate test cases for the program element.

As another example, in code coverage-based fault localization the program elements are ranked according to how suspicious they are to contain the fault based on test case coverage and pass/fail status. Wrong coverage data may influence the fault localization process because if the faulty element is erroneously reported as not covered by a failing test case, the suspicion will move to other (possibly non-defective) program elements.

Besides evaluating the code coverage inaccuracies themselves in an empirical study, this paper also addresses the impact on the mentioned and other applications.

III. Aim of the Study

A. *Code Coverage Measurement for Java*

This paper deals with code coverage measurement for Java programs. Apart from being a popular language in itself, Java coverage measurement is important also because, due to language design and the structure of the bytecode and the virtual machine, measuring code coverage is a relatively simple task (compared to other languages like C++). This is further emphasized by an increased demand for code coverage measurement in agile projects, where continuous integration requires constant monitoring of the code quality and regression testing. This has lead to appearance of a large set of tools for this purpose, many of which are free of charge and open source. However, it seems that the working principles, benefits, drawbacks and any associated risks with these tools are not well understood among practitioners and researchers yet.

In Java, there are two conceptually different approaches for coverage measurement. Common to them is that the system under test and/or the runtime engine is *instrumented*, meaning that "measurement probes" are placed within the system at specific points, which will enable the collection of runtime data, but which do not alter the behaviour of the system. The first approach is to instrument the *source code*, which means that the original code is modified by inserting the probes, then this version is built and executed during testing. The second method is to instrument the compiled version of the system, *i.e.* the *bytecode*. Here, there are two further approaches. First, the probes may be inserted right after the build and effectively producing modified versions of the bytecode files. Second, the instrumentation may take place during runtime upon loading a class for execution. In the following, we will refer to these two approaches as *offline* and *online* bytecode instrumentation, respectively. Some example tools for the three approaches are Clover [21] (source code), Cobertura [22] (offline bytecode) and JaCoCo [23] (online bytecode). There are different possible features available in tools employing these approaches, and they also have various benefits and drawbacks. In Table I, we overview the most important differences (some of them depend on the situation, here we list our subjective assessment).

The many benefits of bytecode instrumentation (*e.g.* easier implementation, no need for source code and separate build) are so attractive that tools employing this technique are far more popular than source code instrumentation-based tools [4]. Furthermore, most users do not take trouble over investigating the drawbacks of this approach and the potential impact on their task at hand. Interestingly, scientific literature is also very poor in this respect, namely systematically investigating the negative effects of bytecode instrumentation on the presentation of results in source code (see Section III-C).

TABLE I
Code Coverage Approaches for Java

Property	Source code	Offline bytecode	Online bytecode
Source code	Needed	Not needed	Not needed
Special runtime	Not needed	Needed	Needed
Bytecode and VM	Not dependent	Dependent	Dependent
Filtering control	Complete	Partial	Partial
Separate build	Yes	No	No
Results in source	Yes	Partially	Partially
Compile time	Impacted	Impacted	Not impacted
Runtime	Impacted	Highly impacted	Impacted
Implementation	Difficult	Easy	Easy

There are important benefits of source code instrumentation visible from the table above, which might overweight bytecode instrumentation in some situations. The most important is that in the situations when the results are to be investigated on source code level (in most of the cases!), mapping needs to be done from computations made on the bytecode level. And due to the fact that perfect one-to-one mapping is not generally possible, this might impose various risks.

B. *Research Questions*

Hence, the primary aim of this research is to start filling this gap and investigate both in quantitative and qualitative terms the following. In what situations and to what extent bytecode instrumentation distorts the code coverage results when they are reflected back to source code, and what is the impact of such inaccuracies on possible applications? To answer this question we conduct an empirical study involving two representative tools – one with source code instrumentation and one with online bytecode instrumentation – and we measure code coverage results on a set of benchmark programs. We then elaborate on the possible causes and impacts.

More precisely, our research questions are:

RQ1 Quantitative evaluation.

- RQ1a How big is the difference between code coverage obtained by the two tools on the benchmark programs?
- RQ1b On a per-method basis, in how many cases is the coverage different?

RQ2 Qualitative evaluation. What are the typical causes for the difference?

RQ3 What is the expected impact of code coverage inaccuracies to a set of applications?

In this paper we calculate and analyze coverage results on *method level*, *i.e.* the basic element of a coverage information is whether a specific Java method is invoked by the tests or not, regardless of what statements or branches are taken in that method. This might seem too coarse a granularity but we think the results will be actionable due to the following. First, in many scenarios coverage analysis is done hierarchically starting from higher level code components like classes and methods. If the coverage result is wrong at this level it will be wrong at lower levels too. One might think that the differences between bytecode and source code instrumentation are more emphasized at statement and branch levels (as other research also showed [5], [24]), however as we will see later in the paper, there are method-level differences in Java which might have significant impact. Finally, method level analysis enables easier measurement of bigger systems as well.

C. Related Work

Despite the importance and the possible risks overviewed above, the differences between bytecode and source code instrumentation have been scarcely systematically investigated. Probably the most closely related work is the study by Li *et al.* [5] in which the authors evaluate the source code and bytecode instrumentation methods and tools from the perspective of *branch coverage* measurement. They used a single program and a part of its test suite to evaluate the coverage tools. They found that source code instrumentation is more appropriate for branch coverage computation which is due to various differences between the generated bytecode and source code. This is aligned with our findings for *method coverage*. Furthermore, our experiments have been performed on more and much bigger systems. Kajo-Mece and Tartari [25] evaluated two coverage tools (source code and bytecode instrumentation based ones) on small programs and concluded that the source code based one is more reliable for the use in determining the quality of their tests.

Alemerien and Magel [24] conducted an experiment in order to investigate how the results of code coverage tools are consistent in terms of line, statement, branch, and method coverage. They chose the overall coverage as the base metric for comparing the tools. Their findings show that branch and method coverage metrics are significantly different, but statement and line coverage metrics are only slightly different. They also found that program size affects significantly the effectiveness of code coverage tools with large programs.

Kessis *et al.* [26] presented a paper in which the authors investigated the usability of coverage analysis from practical point of view. They conducted an empirical study on a large Java middleware application, and found that although some of the coverage measurement tools are not mature enough to handle large scale programs properly, using the adequate measurement policies would radically decrease the cost of coverage analysis, and together with different test techniques it can ensure a better software quality.

There are several papers in which the authors compare different code coverage tools for Java (*e.g.* [27], [28]), but common to these works is that they mostly concentrate on the different features of the tools, and the accuracy of the results they provide are not investigated.

IV. Description of the Experiment

A. Overview

For setting up the experiment for our empirical study, we had two important tasks at first: selecting the code coverage measurement tools (Section IV-B) and the benchmark programs (Section IV-C).

Then, we modified the build system of each subject program to integrate the necessary tasks to collect the coverage data using the two coverage tools. Our extensions of the build and test process included a small modification to ignore the test failures of a module that would normally prevent the compilation of the dependent modules and the whole project. This was necessary when some tests of the project failed on the measured version, and in a few cases when the instrumentation itself caused some tests to fail.

The execution of the tests using the two coverage tools produced coverage data in different forms. Hence, we had to unify various components of the data including the identification of methods, identification and separation of the individual test cases and the representation of the coverage data. For this purpose, we used the SoDA framework [29], [30], which was also able to produce the various analyses on the raw coverage data. Technically, the data generated by the coverage tool was converted into a common SoDA representation – which is essentially a coverage matrix with test cases in its rows and methods in the columns – and then this representation was used to perform the additional analyses.

Note, that this kind of measurement, when method coverage is assigned to individual test cases, will result in the loss of some coverage data. Namely, the coverage of a method is lost if it cannot be directly associated to a specific test case (*e.g.* the method is called at the test class initialization phase). We measured the amount of this loss on our subject programs (see Table II), and found that out of the eight programs it was marginal for three, below 1.2% for two programs, and between 3.8% and 7.3% for the remaining ones. We think that even the higher rates can be accepted in most applications because the lost data represents the coverage of methods that are not intentional goals of the tests (but are technically necessary for the test execution).

B. Tools

For selecting the tools to be used in the experiments, we established the following criteria. First, we aimed at tools that are actively developed and maintained and are popular among users. We measured the popularity of the tool candidates by reviewing technical papers, open source projects, and utilizing our experiences from previous projects. The tools had to handle older and current Java versions including new language constructs (support for at least Java 1.7 but preferably 1.8 was needed) as recent program versions include such constructs. Since we wanted a detailed study about the differences between the tools, we wanted to make sure that we can gather *per-test case* coverage results (*i.e.* which test cases cover each method) from the tools as well. Finally, we wanted the tool to easily integrate into the Maven build system [31], as today this seems to be a popular build system used in many open source projects. In addition, the ability of smooth integration reduces the chances of unwanted change in the behaviour of the system and the tests used in the experiments.

In Section III-A, we discussed three fundamental code coverage calculation approaches for Java. But for the study we chose to use one source code instrumentation tool and one representing the two types of bytecode instrumentation. The reason we do not investigate both types of the latter category is that there are no fundamental differences in how and which program elements are instrumented, only the "timing" of the instrumentation is different.

We selected JaCoCo [23] for the bytecode instrumentation approach. This is a free Java code coverage library developed by the EclEmma team. JaCoCo measurements can be easily integrated into a Maven-based build system, but originally it cannot perform per-test case coverage measurements. So, to be able to gather this information, we slightly modified the execution of the JaCoCo Maven plugin and added a special listener that detected the start and end of the execution of a test case. To validate that this tool will be a good representative of the bytecode instrumentation approach, we performed a one-to-one comparison of coverage data obtained by JaCoCo and Cobertura [22], and found no significant differences.

The tool selected for source code instrumentation approach was Clover [21], a commercial tool by Atlassian. This tool could also be easily integrated in the Maven build process and there was no problems in producing per-test case coverage information. We did a manual verification of the results of Clover by performing manual instrumentation and execution of a subset of our subject systems, and found no deviations from our expected coverage results. Thus, we treat Clover as a "ground truth" for source code coverage measurement.

C. Subjects

For setting up our set of benchmark programs we followed these criteria. As we wanted to compare bytecode and source code instrumentation the source code had to be available. Hence, we used open source projects, which also enables the replication of our experiments. As mentioned, we decided to use the Maven infrastructure in which the code coverage measurement tools easily integrate, so the projects needed to be compilable with this framework. Finally, it was important that the subject programs have a usable set of test cases of realistic size, which are based on the JUnit framework [32] (preferably version 4). This last restriction was necessary because to measure per-test case method coverage the use of this framework was the most straightforward. We searched for candidate projects on GitHub [33], preferring those that had been used in the experiments of previous works. We ended up with 8 subject programs, which belong to different domains and are non-trivial in size (see Table II).

V. Results

A. Total Number of Methods

Before we go into the details of code coverage analysis results, we need to establish a base set of program elements to which the coverage is to be compared. Since we used tools with fundamental differences in the analysis approach, not only the covered methods may differ, but also the total set of methods recognized by the tools. Having the code coverage applications in mind, the basic concept we followed was to use the set of methods which is actually present *in the source code*, and not what is observed at runtime. In this section, we analyze this difference, then we establish the base set.

TABLE II

Subject Programs. The metrics were calculated from the source code (*i.e.* generated code is not counted).

Program	LOC	Methods	Tests	Domain
checkstyle	114K	2 655	1 589	static analysis
commons-lang	69K	2 796	3 678	java library
commons-math	177K	6 835	5 805	java library
joda-time	85K	3 898	4 176	java library
mapdb	53K	1 582	1 784	database
netty	140K	8 133	4 079	networking
orientdb	229K	13 052	1 058	database
oryx	31K	1 557	208	machine learning

1) Test methods: Unit tests themselves should not be investigated for coverage, hence all methods of unit test classes needed to be excluded from further analysis. JaCoCo relies on the project description to determine the test methods. On the other hand, Clover tries to determine test methods by checking the class and method names, and in most cases this is reliable. However, in some cases when test class names did not follow the naming conventions, Clover misclassified tests as regular methods. To correct these errors, we relied on the Maven project hierarchy and examined the source path information of the classes, and we filtered out those methods that were located in the test source directories, *e.g.* `src/test`.

After this initial filtering, the lists of all methods for a program in the two tools differed in both directions: on average, 6.43% of the methods (recognized by either tool) were present only in JaCoCo, 5.42% were recognized only by Clover, and only 88.15% were present in both of them.

2) Generated methods: The difference in favor of JaCoCo consisted of various methods generated by the Java compiler. Generated methods are considered for the coverage analysis by most bytecode instrumentation tools – including JaCoCo, however a source code tool like Clover does not include them, of course. To recognize generated methods, we used a third-party static analyzer (the SourceMeter [34] tool). This tool could extract a lot of information regarding the methods including whether they were generated ones or present in the source code.

Table III shows the total number of generated methods in its last two columns (the percentage is relative to the number of methods in Clover results). Columns 6–8 of the table show what constitute these, including default constructors (if they are not given in the source), "`<clinit>`" methods, and access methods in the case of some nested class operations. The second to fourth columns are provided for reference, and show how are the non-generated methods from the source code divided into regular methods and user provided constructors.

3) Methods not recognized by JaCoCo*:* The other type of difference is when a method is not recognized by the bytecode instrumentation approach. This can happen in the

TABLE III
PERCENTAGE OF SPECIAL METHODS (SHOWN RELATIVE TO Clover ELEMENTS)

Program	Regular methods		User constructors		Generated constructors	<clinit>	Other generated	Total generated	
checkstyle	2 435	91.96%	213	8.04%	139	82	0	221	8.35%
commons-lang	2 552	91.44%	239	8.56%	35	60	14	109	3.91%
commons-math	5 549	81.29%	1 277	18.71%	149	158	70	377	5.52%
joda-time	3 407	87.70%	478	12.30%	12	74	2	88	2.27%
mapdb	1 444	90.31%	155	9.69%	78	43	0	121	7.57%
netty	8 296	86.67%	1 276	13.33%	257	464	112	833	8.70%
orientdb	12 200	90.26%	1 317	9.74%	487	422	124	1 033	7.64%
oryx	1 749	84.29%	326	15.71%	77	80	12	169	8.14%

following situations. A submodule of the program has no tests, so JaCoCo is not executed and cannot collect information about the methods of the module. Or, the signature of the generated method does not match (for technical reasons) the source code signature (*e.g.* it has additional parameters inserted by the compiler). Table IV shows what we measured for this aspect. It shows in the second column how many methods are recognized by both tools, and how many only by Clover (third column). The last column shows the sum of these two values, *i.e.* the total number of methods recognized by Clover. Observe that for the last three programs, the number of unrecognized methods by JaCoCo is quite high, while for the other programs it is 1% of Clover's methods or much fewer. The main reason for this is that these programs are organized into multiple submodules and JaCoCo handles the coverage of submodules differently than Clover. JaCoCo considers a method as covered if it is tested by its own module, while Clover aggregates coverage among all modules.

TABLE IV
NUMBER OF ALL METHODS

Program	Both	Clover only	Total Clover
checkstyle	2 646	2	2 648
commons-lang	2 778	13	2 791
commons-math	6 807	19	6 826
joda-time	3 871	14	3 885
mapdb	1 582	17	1 599
netty	8 133	1 439	9 572
orientdb	13 052	465	13 517
oryx	1 557	518	2 075

Eventually, we established our *base set* to be the total set of methods recognized by Clover, visible in the last column of Table IV. Comparing to this base set we are be able to precisely assess the accuracy of the bytecode instrumentation approach *with respect to the source code*.

The base set supposed to be the set of methods actually appearing in the source code, so we wanted to verify if Clover produces this list accurately. For that we used again the SourceMeter tool, and found that there were no differences between the two lists in any of the programs.

B. Quantitative Evaluation

In this section, we present our measurement data regarding the differences in the coverage reported by JaCoCo and Clover. The qualitative evaluation of the possible differences will be presented in the next section.

1) Total coverage: First, we compared the overall method-level code coverage values obtained by the two tools for each program (this addresses our research question RQ1a). To calculate this, we divided the number of methods reported as covered by each tool by the total number of methods reported by Clover (the base set). In the case of JaCoCo, the number of covered items included only methods which were recognized by both tools, *i.e.* covered generated methods were excluded. We also did not distinguish the cases when a method was not recognized from recognized but not covered by JaCoCo. This way, we achieved the most relevant comparison in terms of the source code.

By treating Clover as the ground truth (with its base set and the covered set, both verified manually and using a third-party tool) and treating unrecognized methods by JaCoCo as not covered, there were only two types of inaccuracies remaining. We will use the term *falsely covering* to denote the situation when JaCoCo reports a method as covered while it should not be (according to Clover); and the term *falsely not covering* when JaCoCo does not cover a method while Clover does cover it.

The overall coverage data can be seen in Table V, where the mentioned numbers and coverage percentages are shown, respectively.[1] The difference is between 0.2% and 8.5%, but what is interesting is that programs netty, orientdb and oryx produced significantly larger difference. Also, except commons-math, JaCoCo's coverage was always smaller. This suggests that bytecode instrumentation typically shows the *safe but imprecise* case, but we investigate these differences in more detail in the following sections.

Table VI shows how big is the difference in the coverage between JaCoCo and Clover *relative* to the coverage obtained by Clover. This measure is relevant because we treat the source code instrumentation results as the ground truth to which bytecode based results should be compared. The last column of the table shows the difference in the number of covered methods overall for the whole test suite (in either direction), divided by the number of methods covered by Clover. As can be seen, the difference is more emphasized for the last three

[1] Note, that the overall percentage shown for JaCoCo is not directly comparable to any percentage that the tool originally reports through its UI due to a different denominator and the filtering mentioned above.

TABLE V
OVERALL COVERAGE RATIOS

Program	Total Clover	Covered JaCoCo	Covered Clover	Percent JaCoCo	Percent Clover
checkstyle	2 648	2 483	2 501	93.8%	94.4%
commons-lang	2 791	2 608	2 615	93.4%	93.7%
commons-math	6 826	5 998	5 989	87.9%	87.7%
joda-time	3 885	3 516	3 532	90.5%	90.9%
mapdb	1 599	1 237	1 243	77.4%	77.7%
netty	9 572	3 552	4 361	37.1%	45.6%
orientdb	13 517	4 233	5 311	31.3%	39.3%
oryx	2 075	434	570	20.9%	27.5%

programs according to this measurement than what we got in Table V. For program oryx, for instance, the number of erroneously reported methods by JaCoCo is nearly one quarter of the number of covered methods by Clover.

TABLE VI
RELATIVE DIFFERENCE IN THE COVERAGE

Program	Per-test case average				Overall
	Average	Q1	Median	Q3	coverage
checkstyle	9.11%	0.00%	0.00%	0.00%	0.719%
commons-lang	6.99%	0.00%	0.00%	1.85%	0.267%
commons-math	40.78%	0.00%	0.83%	15.79%	0.150%
joda-time	34.56%	0.00%	7.59%	10.42%	0.453%
mapdb	243.50%	0.00%	8.36%	96.15%	0.482%
netty	126.95%	19.35%	43.91%	94.32%	18.550%
orientdb	6.25%	0.00%	0.00%	1.90%	20.297%
oryx	100.83%	16.67%	50.00%	100.00%	23.859%

We also performed this kind of measurement individually for each test case in the test suites of the benchmark programs. Namely, we calculated how big is the difference for each individual test case relative to Clover data. The result can be seen in the second to fifth columns of the table, where the average, lower quartile (Q1), median, and upper quartile (Q3) values are shown, respectively. It can be observed that the difference is quite varying and is, not surprisingly, not really related to the overall coverage data. There are, however, some quite high average values such as for mapdb, netty and oryx. This means that, on average, the number of erroneously reported methods by JaCoCo may be as high as more than two times of the number of covered methods by Clover. The median and quartile values show that the high average difference is probably caused by a small number of outliers.

2) Per-Method Coverage: In the next experiment, we recorded for each method from the base set how many of the test cases cover that method according to the two tools. Then we counted the number of methods for which the number of covering test cases was equal, and how many times one or the other tool reported this differently (addressing our research question RQ1b). To compare the "number of covering test cases" we used two approaches: in a less strict one only the coverage fact was compared, *i.e.* if it was covered at least once or not at all. In a more strict approach the two numbers of test cases were directly compared to each other. This enabled us a more subtle analysis of the differences in the overall coverage results from the previous section.

There may be two main kinds of differences in the number of covering test cases: Clover reports a method covered while according to JaCoCo it is not covered, and vice versa (with the strict approach, this is the direction of the non-equal relation). Based on results so far, we expected that there will be no cases when JaCoCo covers a source element while Clover does not cover it. However, we found several cases when this was not true. Altogether there were 116 such problematic methods, which we all investigated manually to find out the reasons for the difference. We found that it was due to various differences related to the behaviour of the tools or some peculiar aspects of our measurement environment. Namely, in 63 cases the method in question was called from test fixture code. Clover uses its own mechanism to identify test cases and decide what covered program elements belong to the individual test cases, and which ones are "common," belonging to none individually. JaCoCo employs a slightly different concept, and treats these common methods as part of each individual test case. The mentioned common test code includes *setup* and *teardown* test methods (annotated by `@Before` or `@After` in JUnit) and test class constructors. In the next 13 cases the methods reported by JaCoCo as covered were invoked during test class initialization (*i.e.* from the generated `<clinit>` method), class setup, or class teardown (annotated by `@BeforeClass` and `@AfterClass` in JUnit). The remaining 40 cases were caused by the deficiency of Clover to recognize a test case as a test case.

In the following, we concentrate on the other type of difference, namely when JaCoCo fails to report a coverage while Clover does this. Table VII shows the associated results. Columns two and three correspond to the less strict approach: how many times the method was not recognized and how many times it has not been covered by any of the test cases, respectively. The last column shows the result for the strict approach, namely how many times the method was covered by at least one test case but this number was smaller than for the Clover case.

TABLE VII
NUMBER OF METHODS WHERE THE JaCoCo COVERAGE WAS SMALLER THAN THE Clover COVERAGE

Program	Not recognized	Zero tests	Fewer than Clover
checkstyle	2	17	266
commons-lang	13	5	131
commons-math	17	7	324
joda-time	12	7	226
mapdb	17	2	30
netty	24	800	1 735
orientdb	72	1 040	2 682
oryx	2	140	91

We can observe the following from the table. First, there are relatively few cases of the inaccuracy of JaCoCo which are due to unrecognized methods. The other thing to notice is that where the difference of the coverage was higher (see Table V), this is also reflected in this table. The strict measurement follows this trend but we can see that even when the coverage fact itself is not impacted there are still many inaccuracies in the details of the coverage results. We elaborate on the possible reasons for this particular set of differences in the next section.

In Figure 1, the summary of differences is shown. This diagram is based on the numbers from the previous table, this time relative to the number of methods in the base set. The results indicate that in the three programs with the biggest overall coverage difference, about 6–8% of the methods are falsely reported as not covered, and that when all factors including not recognized and fewer times covered cases are taken into account, the difference is much more emphasized. On average, 17.9% of all methods of the 8 programs have false coverage data, when investigated more closely.

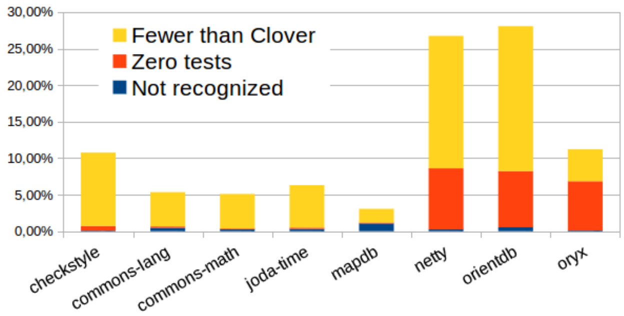

Fig. 1. Summary of differences in the per-method coverage

C. Qualitative Evaluation

In this section, we address our research question RQ2, *i.e.* the possible causes for the differences we observed and presented in the previous section. We carefully examined the differences between the coverage results reported by JaCoCo and Clover by looking at the source code and generated bytecode if necessary, as well as other factors like build configuration. Due to their large number, we could not look into each individual difference, instead we manually selected the typical cases making sure that each system and module is sufficiently covered by our investigation. Altogether, we manually investigated several hundred individual methods and test cases during this work. Finally, we were able to identify a set of common reasons, which we overview in the following.

1) Instrumentation: In some cases the instrumentation itself modifies the behavior of the tests, which may influence the list of executed methods. An example is in the joda-time program, where two specific test cases fail after being instrumented by Clover. This is because the tests depend on a number of subclasses of the tested class, and – as Clover implements coverage measurements and test case detection by inserting subclasses into the examined class – these two tests fail on assertions right at the beginning of the tests. Similar failures occur in the checkstyle project as well, where two of the test cases try to ensure that the classes they test have a fixed number of fields, however with the additional fields which Clover inserts in the classes these assertions fail.

2) Cross-coverage among submodules: Larger systems are organized into submodules, which is also reflected in the build configuration, in our case the Maven projects. Here, a submodule can define its own dependencies and build process including unit tests. This means that each submodule has its own test suite with a set of test cases. JaCoCo and Clover handle submodules differently. Let A and B be two different submodules in a Maven project. During the build (and test phase) of module B, module A is treated as an "external" dependency, which prevents JaCoCo from instrumenting and measuring the coverage of the elements of A. Thus, it only considers a method of module A covered if the method is invoked from the tests of module A. On the other hand, Clover aggregates coverage among all modules, so if a method from A is used in a test in B, Clover considers the method as covered. These different behaviors can lead to differences in the global coverage of the projects. Our subjects netty, orientdb and oryx are examples of multiple module projects, and these have the biggest difference in the coverage as well (see Table V). The other five programs are single module projects. Although we investigated only Maven based projects in our experiments, we think that similar problems may occur in other build configuration systems as well.

3) Untested submodules: The "Clover only" column in Table IV lists the number of methods recognized only by Clover but not recognized by JaCoCo. This difference is another consequence of the different submodule handling between JaCoCo and Clover. In the case of JaCoCo, if module A does not have any tests, it will not recognize any of its methods, which means that the methods of A will be missed from the set of all methods of the project. Clover, on the other hand, correctly determines the set of all methods across all submodules. For example, netty has 1439 methods which are recognized by Clover only. 1395 of these methods are in submodules that do not have any tests. oryx has 518 such methods, 516 of them are in untested submodules, and orientdb has 278 of 465 methods that are not recognized by JaCoCo for the same reason.

4) Name encoding: A common reason for the differences is related to enums, anonymous and nested classes. The problem is that in some cases a method of such a class may get additional parameters to access the members of its enclosing class. Other cases show that methods might even lose some of their parameters. These modifications result in different signatures of the source code and bytecode instance of the same method.

For example, a constructor like `MyEnum(String name)` of an `enum` type in the `pack` package will have the signature `pack/MyEnum/MyEnum(LString;)V` in the source code, while the bytecode based tools will see it as `pack/MyEnum/MyEnum(LString;ILString;)V`. Another example is when there is a private static class named `Bar` with a private constructor `Bar(final Foo f)` nested in a final class named `Foo`. The source code based tools recognize the constructor as `Foo$Bar(LFoo;)V`, while bytecode based ones will see `Foo$Bar()V`.

Such missing or extra parameters in the bytecode make the signatures of these methods different in JaCoCo and Clover measurements. This difference prevents the automatic assignment of the methods of the two measurements, and caused the reduction of JaCoCo coverage count in our experiments.

5) Exception handling during coverage measurement: When JaCoCo instruments the bytecode, it inserts probes into strategic locations by analyzing the control flow of all methods of a class. If the control flow is interrupted by an exception between two probes, JaCoCo will not consider the instructions between the probes as covered. For example, if a method throws an exception at the beginning of the caller method, JaCoCo marks the caller method as not covered. However, Clover's instrumentation strategy is able to handle this situation and it will mark the caller method as covered.

6) Generated code: All programs we investigated include code constructs that result in compiler generated methods, which are not visible in source code, only in bytecode. This includes default constructors and initializers as well, but we handled these cases by the aforementioned filtering at the beginning of the measurements. On the other hand, some projects (for instance, checkstyle) generate some portion of the code of the application on-the-fly using some external tool like ANTLR, or a configuration setting. Instrumentation tools may consider this situation differently, but usually they can be configured to consider also this section of the source code as part of the code base. Bytecode instrumentation tools are able to handle generated code more easily.

7) Other: Although we did not conduct qualitative analysis on other issues we think that there might be other hidden problems, *e.g.* dynamic runtime optimization performed by the JVM, which may affect the differences between JaCoCo and Clover measurements.

To summarize, we found that although there are some tool specific issues, most of them are generalizable, and will probably be applicable to other bytecode and source code instrumentation based tools. Indeed, we found that most of the specific issues of JaCoCo are present in Cobertura, the other bytecode instrumentation tool we considered, as well.

D. Impact on applications

In this section, we address our research question RQ3, *i.e.* what are the possible implications of the inaccuracies of bytecode instrumentation. We collected some of the most important applications of code coverage measurement, which are summarized in Table VIII. The table lists the applications and briefly mentions the impact of both cases of the inaccuracies: the second column shows the case when an actual coverage is not reported by the tool, and the third column is the opposite.

If we compare the different cases we can observe that the level of impact is typically not the same for falsely covering and falsely not covering. In the following, we elaborate on the different applications in more detail.

1) White-box testing, quality, traceability: As mentioned at the beginning of the article, white-box testing may suffer from coverage inaccuracies in two ways. The falsely covering case is a more serious one because it may give a false confidence in the completeness of the testing. Fortunately, our results show that bytecode instrumentation rarely results in this kind of error. However, the other case is much more frequent, as presented in the previous section. Here, falsely not covering program elements will usually result in more effort required to action on the coverage results. Namely, it will mean more program elements to investigate during testing. Results from Figure 1 show that, for instance, when investigating uncovered methods of the last three programs about 6–8% of the methods will be examined superfluously.

TABLE VIII
SUMMARY OF THE IMPACTS OF INACCURATE CODE COVERAGE

Application	*Falsely not covering*	*Falsely covering*
White-box testing, quality, traceability	Increased effort	False confidence
Fault localization	Impossible localizability	Moderate impact on scores
Test selection and prioritization	Suboptimal priority	Suboptimal priority
Test case generation	Inability to check success	Reduction in success
Mutation analysis	Minor	Inability to kill mutants

2) Fault localization: Code coverage based fault localization (sometimes referred to as spectrum based fault localization) [35], [16] fundamentally relies on code coverage. There are variations to it, but the basic approach of this technique is to narrow down the search for the faulty program element based on a "suspiciousness score." It will be higher for a program element if there are many test cases that cover it and fail, while test cases not covering it pass at the same time. Clearly, an error in the coverage will impact the suspiciousness score and hence the chances of localizing the fault. In particular, if the fault is in the program element which is erroneously reported as not covered it will never be localized using the standard algorithms (most scores such as Tarantula [12] will be set to 0 in this case). Even in the case when the coverage is not totally missing but there are fewer covering test cases reported (the strict approach from Section V-B2), it will decrease the chances for fault localization. For instance, for some of the programs in Figure 1 up to 25% of the methods will be affected by this issue. The falsely covering case will also impact the localization scores, though moderately.

3) Test selection and prioritization: Test selection and prioritization methods that rely on code coverage ([15], [16], [17]) may also be severely impacted by inaccuracies in the coverage. Algorithms that give preference to highly covering test cases basically prioritize test cases either globally according to the coverage ratio or to how much additional coverage a test case provides [36]. Test selection methods then select the first given number of test cases from this list. This means that any difference in the *per-test case* coverage will have a high impact on the performance of the algorithms. In Section V-B1, we have seen that the difference in per-test case coverage may be quite dramatic: from 100% to nearly 250% of the number of correctly covered methods will appear as incorrect coverage information for some of the programs. This can then fundamentally influence the priority lists computed by the algorithms. This problem affects both the falsely not covering and falsely covering cases.

4) Test case generation: The impact of inaccurate code coverage on test case generation algorithms [10], [11] is similar to the impact on general white-box test design technique. Except that in this case there is no human involved who can understand (with increased effort) that there is a missing or superfluous coverage. Consequently, a missing coverage will result in the algorithm not being able to check the success of a generated test case because it will always observe that it failed to cover a program element. A superfluous coverage on the other hand, will reduce the successfulness of the generation because the algorithm will think that an element is covered by some of the generated test cases and will not continue searching further.

5) Mutation analysis: Our final example of code coverage application is mutation analysis and mutation testing [18], [19]. In mutation analysis, the program is modified to insert artificial faults (creating *mutants*) and verify if any of the test cases can detect the fault (in other words, can *kill* the mutant). If the mutant is not killed, then in a mutation testing approach a new test case is created or generated to kill it. Here, the falsely covering case will have the effect that the algorithm will be unable to kill the mutant because due to actual non-coverage the fault will not be detected (only false belief will remain due to the reported coverage). A falsely not covering case will have a minor impact because the fault will be detected regardless of code coverage information, but it will still be confusing. Also, mutation testing will suffer from the same difficulties as with test case generation since it uses this technique to augment the test cases in order to kill the mutants.

VI. Threats to Validity

The main aim of the paper was to investigate the effects of bytecode instrumentation technique for code coverage measurement with respect to source code instrumentation. We did this by empirical measurements using two specific tools. Clearly, this raises the question how generalizable are the result to other tools using similar techniques. When interpreting the results, we tried to separate tool-specific issues from approach-specific ones. Also, we did a comparison of bytecode instrumentation results of JaCoCo to that of a similar tool, Cobertura and found no significant differences. Finally, the results of source code instrumentation with Clover have been verified with manual instrumentation.

Our experiments showed results with respect to method-level coverage analysis. Generalization to other granularities such as statement or branch level may not directly be possible. We used a limited set of Java programs, and our findings may not be valid for other programs. Furthermore, all projects use the Maven build system and JUnit test framework, which could also affect the generizability of the results. However, these programs were selected from different domains and have different size (both in terms of code and tests).

We slightly modified the instrumentation process of JaCoCo by adding a listener that detected the start and end of the execution of a test case. The results obtained with this modification may not directly translate to coverage results everyday users would experience with the stock version of JaCoCo. However, we compared the results of the unmodified JaCoCo measurement to our version in terms of actually covered program elements and found no significant differences.

VII. Conclusions and Future Work

Our experiments presented in this paper show that the difference between bytecode and source code instrumentation based code coverage measurement tools can be significant. To our knowledge, this kind of systematic comparison with realistic systems and test suites has not been presented previously. At first, the difference in the overall percentage results presented to the user might not seem too big (we found it to be between 0.5–8.5%), but relative to the actually covered elements, the difference can be as high as 24%. The difference might not influence an engineer's decisions in some situations, *e.g.* when the overall confidence in the testing is to be assessed. However, on a per-test and per-method levels, it can have bigger impact. In the last section we listed several applications where these fine differences matter a lot.

In this research, we used two specific tools that are popular in industry and research: Clover and JaCoCo. They represent source code and bytecode instrumentation approaches, respectively. Although we found some tool-specific differences, we think that most of our findings will be applicable to other tools with similar working principles.

The implications of the results might be multiple. For industry practitioners, it will depend on the actual application of code coverage measurement but it will range from increased effort to false confidence in testing. Apart from the numerous advantages of bytecode instrumentation, one should consider also the drawbacks we identified. In academic research, the inaccuracies of the tools used may have great impact on the validity of the results, so we think that, depending on the application, usage of source code level instrumentation should be considered. Finally, our list of possible reasons for the difference may be used as guidelines how to avoid and workaround the inaccuracies if bytecode level instrumentation tools are still to be used.

In this work, we concentrated on method-level analysis, but we plan to extend the experiment to other coverage criteria such as statement and branch. We plan to consider other tools for more careful analysis as well such as offline bytecode instrumentation tools. Finally, in this phase we only considered the potential impact on the applications but did not actually studied it empirically. In the near future, we will consider the actual measurable impacts of the inaccuracies to some of the applications mentioned in this paper.

Online appendix

The measurement data are available online at
http://www.sed.inf.u-szeged.hu/java-instrumentation

References

[1] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, pp. 33:1–33:11.

[2] J. M. Graham Bath, *The Software Test Engineer's Handbook: A Study Guide for the ISTQB Test Analyst and Technical Analyst Advanced Level Certificates, 2nd ed.* Rocky Nook, 2014.

[3] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

[4] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[5] N. Li, X. Meng, J. Offutt, and L. Deng, "Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report)," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, Nov 2013, pp. 380–389.

[6] T. Ostrand, "White-box testing," *Encyclopedia of Software Engineering*, 2002.

[7] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel, "The impact of concurrent coverage metrics on testing effectiveness," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, March 2013, pp. 232–241.

[8] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 435–445.

[9] "The SonarQube open source project," http://www.sonarqube.org/, last visited: 2015-11-13.

[10] S. Rayadurgam and M. Heimdahl, "Coverage based test-case generation using model checkers," in *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the*, 2001, pp. 83–91.

[11] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419.

[12] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. of International Conference on Automated Software Engineering*. ACM, 2005, pp. 273–282.

[13] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, p. 19, 2013.

[14] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *In Proc. Twelfth Int'l. Conf. Testing Computer Softw*, 1995, pp. 111–123.

[15] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 184–208, Apr. 2001.

[16] L. Vidács, Á. Beszédes, D. Tengeri, I. Siket, and T. Gyimóthy, "Test suite reduction for fault detection and localization: A combined approach," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, Feb 2014, pp. 204–213.

[17] D. D. Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 371–396, 2015.

[18] M. P. Usaola and P. R. Mateo, "Mutation testing cost reduction techniques: A survey," *IEEE Software*, vol. 27, no. 3, pp. 80–86, 2010.

[19] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[20] A. Perez and R. Abreu, "A diagnosis-based approach to software comprehension," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 37–47. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597151

[21] "Clover java and groovy code coverage tool homepage," https://www.atlassian.com/software/clover/overview, last visited: 2015-11-03.

[22] "Cobertura java code coverage utility homepage," http://cobertura.github.io/cobertura/, last visited: 2015-11-03.

[23] "JaCoCo java code coverage library homepage," http://eclemma.org/jacoco/, last visited: 2015-11-03.

[24] K. Alemerien and K. Magel, "Examining the effectiveness of testing coverage tools: An empirical study." *International Journal of Software Engineering and Its Applications*, vol. 8, no. 5, pp. 139–162, 2014.

[25] E. Kajo-Mece and M. Tartari, "An evaluation of java code coverage testing tools." in *BCI (Local)*, 2012, pp. 72–75.

[26] M. Kessis, Y. Ledru, and G. Vandome, "Experiences in coverage testing of a java middleware," in *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, ser. SEM '05. New York, NY, USA: ACM, 2005, pp. 39–45.

[27] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.

[28] R. Lingampally, A. Gupta, and P. Jalote, "A multipurpose code coverage tool for java," in *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. IEEE, 2007, pp. 261b–261b.

[29] D. Tengeri, Á. Beszédes, D. Havas, and T. Gyimóthy, "Toolset and program repository for code coverage-based test suite analysis and manipulation," in *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*, Sep. 2014, pp. 47–52.

[30] "SoDA library," http://soda.sed.hu, last visited: 2015-11-03.

[31] "The Apache Maven project homepage," https://maven.apache.org/, last visited: 2015-11-12.

[32] "JUnit java unit test framework homepage," http://junit.org/, last visited: 2015-11-03.

[33] "GitHub homepage," https://github.com/, last visited: 2015-11-03.

[34] "SourceMeter Static Source Code Analyzer Homepage," https://www.sourcemeter.com/, last visited: 2015-11-09.

[35] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007, pp. 89–98.

[36] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.